International Journal of Science and Engineering Applications
Volume 13-Issue 03, 30 - 35, 2024, ISSN:- 2319 - 7560
DOI:10.7753/IJSEA1303.1007

Evaluative Comparison of ASGI Web Servers: A Systematic Review

Anton Novikau
Head of Mobile Development
Talaera, 28 Liberty Street, 6th Floor
New York, USA

Abstract: This paper aims to research major ASGI (Asynchronous Server Gateway Interface) server implementations, including Uvicorn, Daphne, and Hypercorn, and provide a comprehensive comparison. Utilizing a methodology that incorporates both quantitative performance metrics and qualitative feature analysis, this study offers a thorough evaluation of these servers in the context of modern asynchronous web applications.

In this study, performance metrics such as request handling capacity (RPS) and resource consumption (specifically, RAM consumption) are carefully measured under controlled conditions to identify the most efficient ASGI implementations. Additionally, we examine the features provided by each ASGI server implementation, covering critical aspects such as protocol support, scalability options, and an overview of license restrictions.

As a result, this study compares the strengths and limitations of each ASGI implementation. Most importantly, it provides valuable insights for developers and system architects in selecting the most suitable ASGI server for their specific needs.

Keywords: Python, Software, Network, ASGI, Web server, Web server performance, Concurrent request handling, Scalability in web servers, Asynchronous web technologies

1. INTRODUCTION

1.1 What is an application server in Python?

An application server, specifically in the context of Python web development, is a type of software that provides an environment where web applications can be executed. They are responsible for handling the low-level details of client request processing. The application server serves as a bridge between the user's client (or browser) and the backend logic of a web application. Tasks managed by these application servers include managing incoming connections, executing application code, providing an effective way of handling connections, scaling, and ensuring security.

Application servers are necessary due to their ability to simplify and manage the complex interactions in client-server systems. They allow developers to focus on building the core functionality of their applications without worrying about the underlying network protocol implementation details.

In the Python ecosystem, there are two major protocols for such servers: WSGI (Web Server Gateway Interface) and ASGI (Asynchronous Server Gateway Interface). WSGI is an older protocol designed for synchronous Python web applications, while ASGI was created for asynchronous applications, allowing them to handle long-lived connections like WebSockets or HTTP polling more efficiently. The choice between these protocols and their corresponding implementations is determined by the application's specific needs, whether it requires handling real-time data, the expected traffic load, and the nature of the tasks it performs.

1.2 History of ASGI

ASGI, the Asynchronous Server Gateway Interface, represents a significant advancement in Python's web development capabilities, especially in the context of building asynchronous web servers. It provides a solution to the limitations of the Web Server Gateway Interface (WSGI), an older Python standard established in 2003 [1]. WSGI, built on a traditional synchronous request-response model, cannot effectively handle communications outside this format. For example, implementing HTTP long-polling, or any technique requiring long-lived connections, poses challenges (though not insurmountable) due to the complexities and limitations of Python's multithreading system. Furthermore, WSGI is incompatible with WebSockets, a popular protocol for asynchronous message exchanges between clients and servers.

ASGI was developed considering these limitations and addresses them. The introduction of new asynchronous features in CPython version 3.5 [2] made the approach used by ASGI feasible. ASGI's design enables a more flexible communication paradigm, allowing servers and clients to exchange information asynchronously once a connection is established [3].

ASGI servers are gaining more traction within the Python community. Modern frameworks, such as Starlette, LiteStar, and Django, have been developed with ASGI in mind or have recently integrated support for it.

1.3 Overview of ASGI & Comparison with WSGI

The Asynchronous Server Gateway Interface (ASGI) by itself is just a specification that could be implemented by protocol servers/application servers. These implementations are utilized by the applications. The server is responsible for

managing low-level implementation – work with sockets, generating connections, and connection-specific events. For each connection, the server calls the application once - after that, the application takes care of the connection's event messages as they happen and produces events back when necessary.

While the general design is similar to WSGI, it has a very distinct attribute - ASGI applications are asynchronous callables. While WSGI only allows an application to receive a single input stream and return a single result before finishing connection, ASGI lets applications receive and send asynchronous event messages as long as the connection is alive.

The protocol server specifies the following interface for client applications (such as web frameworks): it's a coroutine callable (a Python object that implements the __call__ method) that takes three arguments:
'scope' - a dictionary that contains connection scope information. It's guaranteed to contain a 'type' key that defines the connection protocol. It could be 'http', 'websocket', or any other [4].
'receive' - an awaitable callable, it yields new information (event) when it becomes available. It could be an HTTP body or a new WebSocket message sent by the client.
'send' - an awaitable callable, that takes an event as an argument. Event content is defined by the protocol, it could be an HTTP response body or WebSocket message.
An important part of ASGI is its compatibility with WSGI. ASGI servers can host WSGI-based applications by wrapping them through a translation layer that converts ASGI interface into WSGI and back.

1.4 WSGI Limitations

While Python is widely used in web development, it has a limitation that dramatically limits its potential performance - the Global Interpreter Lock (GIL). The GIL is a part of CPython's memory management mechanic, it's a global mutex that doesn't allow native threads to execute Python bytecode simultaneously. CPython is the most popular implementation of a Python interpreter, directly supported by the Python foundation. This lock was put in place because CPython's memory management is not thread-safe. Since GIL prevents bytecode from running simultaneously in multiple threads on multiple CPU cores it can lead to performance bottlenecks in multi-threaded applications, undermining the potential benefits of parallelism on multi-core processors.
Outside of Python, the most popular design option for web servers to run multiple requests simultaneously is by having multiple workers, where most commonly, workers are native threads that run simultaneously in the same memory space and allow web servers to handle requests concurrently without major impact caused by need in interprocess communication[8]. Another weak spot of WSGI is that its design is not suited for non-blocking IO calls. While many WSGI web servers utilize non-blocking IO internally (see uWSGI), generally, WSGI design prohibits applications from yielding control to the web server while they await for non-blocking operations to finish. Thus, it is usually impossible to handle other requests in the same worker while the application waits for non-blocking API calls to complete. It could be mitigated by using Gevent, which patches IO calls and replaces them with non-blocking versions. It comes with a cost, since Gevent can't patch all of the IO calls, especially made in unsupported packages, additionally, not many WSGI servers support Gevent. With the GIL in place and given WSGI's constraints, the most popular option to support running multiple requests simultaneously is to run web servers with multiple processes, which is significantly more resource-expensive than running them using threads.
ASGI solves this issue by utilizing Python coroutines and allowing its implementations & applications to take advantage of advanced features such as non-blocking IO. By using a non-blocking API, web servers can handle hundreds or thousands of simultaneous connections/requests, which is challenging for WSGI-based applications. The very design of ASGI forces developers to utilize coroutines.

1.5 Objective

Currently there are many implementations of ASGI protocol servers, 3 most popular (according to their rating on Github.com) are:
uvicorn - the most widely used ASGI web server, currently supports only the HTTP/1.1 and WebSockets protocols[5].
daphne - is one of the first ASGI implementations. It was developed for the Django project, specifically for Django channels.
hypercorn - ASGI web server, that supports HTTP/1, HTTP/2 and WebSockets (over HTTP/1 and HTTP/2) protocols.
Each web server evaluated in this research fully supports the ASGI protocol and offers integration capabilities with ASGI web applications. Presently, there exists an absence of definitive guidance for developers regarding the selection among these distinct ASGI implementations. This study is designed to methodically investigate the differences among the three principal ASGI servers—focusing on a comprehensive evaluation employing quantitative performance metrics alongside qualitative feature analysis. The aim is to equip software engineers, software architects, and solution architects with definitive guidelines delineating the advantages and limitations inherent in each option. By elucidating the strengths and weaknesses of each server, this study intends to aid in the selection of the most suitable server for specific applications, thereby enhancing the efficiency, reliability, and overall user experience in web application development.

2. RELATED LITERATURE

The topic of measuring software performance, and specifically web servers, is well studied. For example, Radhakrishnan and John [9] researched a difference between serving static and dynamic data by web servers in a controlled environment. In their 2011 work, Abbas and Kumar study the performance of Web Servers as perceived by clients. They focused on 2 scenarios: when there is no data flow between Web Server and client, and when there is a data flow from web server to client [10].
Ehrlich, Hariharan, Reeser and Mei (2001) proposed an end-to-end performance model for Web-servers in distributing computing and predictions produced by the model matched the performance measured in the test environment.

International Journal of Science and Engineering Applications
Volume 13-Issue 03, 30 - 35, 2024, ISSN:- 2319 - 7560
DOI:10.7753/IJSEA1303.1007

Additionally, Kunda, Chihana and Sinyinda managed to compare popular web servers, including Apache, Nginx and IIS. They came to the conclusion that Nginx outperforms most web servers in many metrics, including CPU utilization, response time and memory usage. [11]
It's worth mentioning that this study should be viewed in the context of development using Python language. There are multiple valuable researches that focus on performance issues in Python, such as work by Ziogas, Schneider, Ben-Nun, Calotoiu, Matteis, Licht, Lavarini and Hoefler. They present a workflow, that both keeps Python's high productivity and achieves portable performance across different architectures.[12]

3. METHODOLOGY

One of the critical criteria for web server ranking is their performance rating. Performance depends on many attributes, most importantly on the implementation details of the web servers in question. A performance comparison will be performed by creating an ASGI-compatible application with a set of endpoints. Some of these endpoints will behave similarly to a regular HTTP-based application, including making asynchronous calls to the database, authenticating users, and generating a JSON response to be returned by the web server.

```
async def get_current_user_email(token: str = Depends(oauth2_scheme), user_db: aioredis.Redis = Depends(user_redis)) -> str:
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        email: str = payload.get("sub")
        if email is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    pwd = await user_db.get(email)
    if pwd is None:
        raise credentials_exception
    return email


new *
@app.get( path: "/api/events", response_model=List[str])
async def get_activity(current_user: str = Depends(get_current_user_email), message_db: aioredis.Redis = Depends(message_redis)):
    items = await message_db.lrange(current_user, -10, -1)
    return list(items)
```

Figure. 1 Test endpoint that authenticates user, utilizes database and returns JSON response

There are also two endpoints that return a static plain text response. One of them returns a response with a body size of 22 bytes, and the other returns a response with a body size of 430 kb. Both endpoints are useful for comparing the "clean" performance of the web servers – one not influenced by possible side effects of having a DB connection – and how well these servers can act as static servers.
Testing will be conducted using a performance load testing tool called "locust," configured to run 6 worker processes that perform simultaneous calls to the defined endpoints. The target metrics for testing include requests per second (RPS) and memory consumption. The requests per second metric is provided by locust, while memory consumption is tracked using a Python script that collects the memory footprint of all the processes belonging to the web server. Choosing requests per second (RPS) and memory consumption as the primary metrics for comparing ASGI implementations is grounded in their direct relevance to web server performance and scalability in real-world applications.
RPS is a direct indicator of a server's ability to manage incoming traffic. High RPS values suggest that the server can handle a larger number of simultaneous requests, making it suitable for high-traffic applications. From the end-users perspective, high RPS often means faster response times, which leads to better user experience.
Memory consumption measures how much RAM a server uses under various loads. It's crucial to make sure that memory is used efficiently, as it's a major factor in maintaining system stability and ensuring that applications can run without exhausting memory resources.
All tests were performed on a MacBook Pro 2019 with a 2.6 GHz 6-Core Intel Core i7 CPU and 16 GB RAM. Versions of software used in these tests are: Hypercorn - 0.16.0, Uvicorn - 0.27.0.post1 and daphne - 4.0.0.
Regarding the different worker classes supported by Hypercorn, separate tests have been performed, and the default configuration of Hypercorn (running with asyncio) shows the same results as uvloop-based workers and much better results than trio-based workers.
The comparative analysis of supported protocols and built-in scalability is a critical factor in the evaluation of ASGI server implementations, often surpassing raw performance metrics in importance. The reason for this prioritization originates from the fact that the ability of a server to support a wide range of protocols, such as HTTP/2 and WebSocket can be pivotal for the development and deployment of modern, real-time web applications. These protocols facilitate efficient, bi-directional communication between clients and servers, enabling the creation of highly interactive and responsive user experiences.
Furthermore, built-in scalability mechanisms, including the support for multiple workers and the ability to seamlessly integrate with load balancers, are essential for applications that must scale in response to varying load. Such mechanisms ensure that an application can maintain high performance and availability, even under significant traffic by distributing the load across multiple instances or processes. Given above, this work includes comparison of supported protocols and scalability mechanics.
Another important metric discussed in this study is license limitations of each ASGI server. License can define when usage of the project is allowed or prohibited, therefore an analysis and comparison of licenses is included.

4. EXPERIMENT

4.1 Comparing static endpoints performance

In order to test the case when a server serves static data, two endpoints were implemented. The "GET /api/longbody" endpoint returns a plain text body of 430 kilobytes in size, and the "GET /api/shortbody" endpoint returns a plain text body of 22 bytes in size. This configuration allows us to test ASGI implementations in different scenarios.

International Journal of Science and Engineering Applications
Volume 13-Issue 03, 30 - 35, 2024, ISSN:- 2319 - 7560
DOI:10.7753/IJSEA1303.1007

Table 1. ASGI servers performance metrics for "GET /api/longbody" endpoint

Name	Requests per second	Memory (RSS) consumption MB
uvicorn	2460	80,8
daphne	1039	110,5
hypercorn	2021	116,5

Table 2. ASGI servers performance metrics for "GET /api/longbody" endpoint

Name	Requests per second	Memory (RSS) consumption MB
uvicorn	4253	79,4
daphne	3073	126,7
hypercorn	3112	112,8

From the data presented in these tables, it's clear that the performance of the servers degrades when returning larger response bodies, although this does not significantly impact memory consumption.
In scenarios involving the serving of short responses, Uvicorn distinguishes itself by outperforming its closest competitor by 36% in terms of the number of requests served per second. Hypercorn and Daphne exhibit comparable performance when managing small data loads, yet there is a twofold difference in their request handling capabilities when dealing with larger data volumes. Regarding memory usage, Uvicorn consistently shows the lowest memory consumption in both scenarios, whereas Daphne and Hypercorn demonstrate similar levels of memory utilization.

4.2 Comparing close-to-real-life performance

Table 2. ASGI servers performance metrics during test imitating close-to-real-life flow

Name	Requests per second	Memory (RSS) consumption MB
uvicorn	1511	77,6
daphne	1258	108,7
hypercorn	1230	109,4

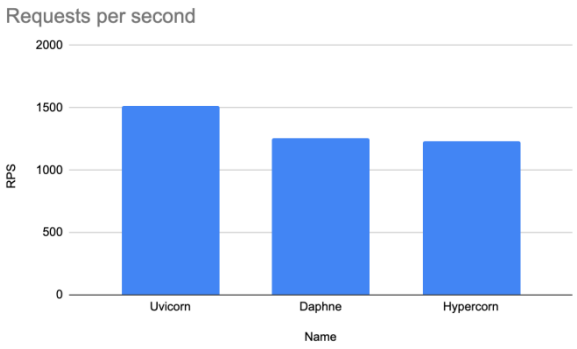

Figure. 2 Test endpoint that authenticates user, utilizes database and returns JSON response

Upon a detailed analysis of the performance metrics, it is clear that Uvicorn, Daphne, and Hypercorn display remarkably similar performance profiles when subjected to tests that closely mimic real-world conditions. Although Uvicorn is approximately 20% faster than both Hypercorn and Daphne, this advantage, when converted into absolute numbers, seems to be relatively minor.
Further review reveals Uvicorn has demonstrated significantly lower memory consumption in all of the tested scenarios.This efficiency in resource usage can be particularly advantageous in environments where memory resources are limited or when running multiple applications simultaneously, offering potential cost savings and enhanced application scalability. Along with exceptional performance, memory usage profile of Uvicorn only contributes to its performance excellence and makes it a compelling choice for developers seeking an optimal balance between speed and resource management in their ASGI applications.

4.3 Comparing supported protocols

While Uvicorn shows exceptional performance, it has a very limited number of supported protocols. At this moment Uvicorn only supports HTTP 1.1 and Websockets. On the other hand, Daphne is capable of handling HTTP, HTTP2, and WebSocket protocols. Daphne's support for HTTP2 makes it distinct from Uvicorn - HTTP2 improves performance of web applications by using TCP connections more efficiently. This makes Daphne better at support of modern web development practices. Daphne is also a part of the Django ecosystem, which is a great advantage for developers planning to use that framework due to its native integration with Django tools.
Compared to its competitors, Hypercorn has the broadest protocol support, which includes HTTP/1, HTTP/2 and WebSocket. It's also worth highlighting that Hypercorn supports QUIC - an emerging standard designed by Google with the purpose of making the web faster by reducing connection establishment time and improving congestion control[13]. Additionally, Hypercorn boasts support for HTTP/3 through plugins, which allows it to provide better performance in high-latency networks, such as mobile networks.[14]. This makes Hypercorn a solid choice for projects that prioritize long life-time, since it's already prepared to handle next-generation internet protocols.

4.4 Licenses

Project license is a very important metric, since it defines when and how a project could actually be used. Uvicorn and Daphne are licensed under the BSD-3-Clause license. This BSD license is known for its permissiveness, it allows almost unrestricted freedom to use, modify and distribute licensed software. It only requires that all copies of the licensed software include the original copyright notice and a list of conditions. BSD-3-Clause also prohibits the use of the names of the project authors and contributors to promote products created using this software without prior written permission[6].This makes Uvicorn & Daphne attractive options for both commercial and open-source projects, offering flexibility for developers and protection for contributors.

Hypercorn follows the MIT license - another highly permissive free software license. Just like the BSD license, the MIT license allows the software to be used, copied, modified, published, distributed, and/or sold freely. The only significant condition of the MIT license is that all copies of the software must include the original copyright notice and permission notice[7]. The simplicity and permissiveness of the MIT license encourages the widespread use of Hypercorn in both open-source and proprietary projects, facilitating innovation and collaboration.By comparing all three ASGI-implementations, it's safe to conclude that there's no major difference between them in terms of licensing and each of them could be used safely and freely in applications.

4.5 Comparing scalability options

In other programming languages, most of the web servers are scaled by running more OS threads, but because of CPython's multithreading constraint, this is not a viable option. ASGI's design promotes web server implementations to use asynchronous I/O operations, which allows handling multiple connections concurrently in a single thread. Given this, if the server doesn't handle the workload, we're left with process-based scaling through the utilization of worker processes. Uvicorn and Hypercorn both provide built-in support for worker processes, which allows developers to spread the workload across multiple CPU cores and achieve near-linear scalability, limited mostly by the number of CPU cores available.

In contrast, Daphne does not include this multi-worker support as a native feature. It may appear as a limitation; however, there's still a way to scale the Daphne web server. By utilizing a load balancer - a software distributing network requests among multiple nodes - Daphne could handle requests simultaneously in separate processes. It requires additional setup, which is unnecessary for Uvicorn and Hypercorn, but it also introduces the potential for advanced traffic management strategies.

To conclude, Uvicorn and Hypercorn both provide built-in options to scale servers using worker processes, while Daphne doesn't, although it's not impossible to scale Daphne too.

5. CONCLUSIONS

In conclusion, this article offers a comprehensive analysis of ASGI server implementations. Daphne, Hypercorn, and Uvicorn were evaluated in terms of performance, supported protocols, and scalability options. Performance evaluation was conducted through a detailed study of their performance in various real-life scenarios, including serving static data and simulating real-life API endpoints that perform database queries for each incoming request.

It was demonstrated that Uvicorn has superior performance, low resource consumption, and provides a scaling mechanism, which makes it a great choice for high-load scenarios, although it lacks protocol support.

Hypercorn and Daphne, while demonstrating similar performance metrics, have significant differences in other dimensions. Hypercorn comes with support for advanced protocols like QUIC, which makes it a superior option by this criterion. While its overall performance is similar to Daphne's, Hypercorn offers support for scaling by spawning additional worker processes, which puts it at an advantage over Daphne.

Daphne itself is a great choice for Django projects - it was created primarily to be an application server for Django, specifically for Django Channels. It has excellent compatibility with Django and provides integration with the Django CLI.

In terms of licenses, no significant differences were found between these three ASGI implementations.

Developers should choose ASGI servers for their projects based on the specific needs of the projects. This study provides a great explanation of the capabilities and limitations of each ASGI server, which should guide software engineers in their search for the most suitable solution.

6. REFERENCES

[1] PEP 333 – Python Web Server Gateway Interface V1.0 | peps.python.org. (n.d.). https://peps.python.org/pep-0333/

[2] PEP 492 – Coroutines with async and await syntax | peps.python.org. (n.d.). https://peps.python.org/pep-0492

[3] Introduction — ASGI 3.0 documentation. (n.d.). https://asgi.readthedocs.io/en/latest/introduction.html

[4] ASGI (Asynchronous Server Gateway Interface) Specification — ASGI 3.0 documentation. (n.d.). https://asgi.readthedocs.io/en/latest/specs/main.html#specification-details

[5] Encode. (n.d.). GitHub - encode/uvicorn: An ASGI web server, for Python. GitHub. https://github.com/encode/uvicorn

[6] The 3-Clause BSD license. (2023, June 15). Open Source Initiative. https://opensource.org/license/bsd-3-clause

[7] MIT. (n.d.). https://www.mit.edu/~amini/LICENSE.md

[8] Menascé, D. A. (2003). Web server software architectures. IEEE Internet Computing, 7(6), 78-81. https://doi.org/10.1109/MIC.2003.1250588

[9] Radhakrishnan, R., & John, L. K. (1999). A performance study of modern web server applications. In P. Amestoy,

et al. (Eds.), Euro-Par'99 Parallel Processing (Lecture Notes in Computer Science, vol. 1685). Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48311-X_29

[10] Abbas, A.M., Kumar, R. (2011). A Client Perceived Performance Evaluation of Web Servers. In: Abraham, A., Lloret Mauri, J., Buford, J.F., Suzuki, J., Thampi, S.M. (eds) Advances in Computing and Communications. ACC 2011. Communications in Computer and Information Science, vol 191. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-22714-1_32

[11] Kunda, D., Chihana, S., & Sinyinda, M. (2017). Web Server Performance of Apache and Nginx: A Systematic Literature Review. Computer Engineering and Intelligent Systems, 8, 43-52.

[12] Ziogas, A., Schneider, T., Ben-Nun, T., Calotoiu, A., Matteis, T., Licht, J., Lavarini, L., & Hoefler, T. (2021). Productivity, Portability, Performance: Data-Centric Python. SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, 1-15. https://doi.org/10.1145/1122445.1122456

[13] Shreedhar, Tanya & Panda, Rohit & Podanev, Sergey & Bajpai, Vaibhav. (2021). Evaluating QUIC Performance over Web, Cloud Storage and Video Workloads. IEEE Transactions on Network and Service Management. PP. 1-1. 10.1109/TNSM.2021.3134562

[14] Perna, Gianluca & Trevisan, Martino & Giordano, Danilo & Drago, Idilio. (2022). A first look at HTTP/3 adoption and performance. Computer Communications. 187. 10.1016/j.comcom.2022.02.005.